TMC-67

A Parallel Indexed Algorithm for Information Retrieval

Craig Stanfill, Robert Thau, and David Waltz
Thinking Machines Corporation
245 First Street, Cambridge MA 02142

ABSTRACT

DR90-2

In this paper we present a parallel document ranking algorithm suitable for use on databases of 1–1000 GB, resident on primary or secondary storage. The algorithm is based on inverted indexes, and has two advantages over a previously published parallel algorithm for retrieval based on signature files. First, it permits the employment of ranking strategies which cannot be easily implemented using signature files, specifically methods which depend on document–term weighting. Second, it permits the interactive searching of databases resident on secondary storage. The algorithm is evaluated via a mixture of analytic and simulation techniques, with a particular focus on how cost–effectiveness and efficiency change as the size of the database, number of processors, and cost of memory are altered. In particular, we find that if the ratio of the number of processors and/or disks to the size of the database is held constant, then the cost–effectiveness of the resulting system remains constant. Furthermore, for a given size of database, there is a number of processors which optimizes cost–effectiveness. Estimated response times are also presented. Using these methods, it appears that cost–effective interactive access to databases in the 100–1000 GB range can be achieved using current technology.

A Parallel Indexed Algorithm for Information Retrieval

Craig Stanfill, Robert Thau, and David Waltz
Thinking Machines Corporation
245 First Street, Cambridge MA 02142

ABSTRACT

In this paper we present a parallel document ranking algorithm suitable for use on databases of 1-1000 GB, resident on primary or secondary storage. The algorithm is based on inverted indexes, and has two advantages over a previously published parallel algorithm for retrieval based on signature files. First, it permits the employment of ranking strategies which cannot be easily implemented using signature files, specifically methods which depend on document-term weighting. Second, it permits the interactive searching of databases resident on secondary storage. The algorithm is evaluated via a mixture of analytic and simulation techniques, with a particular focus on how cost-effectiveness and efficiency change as the size of the database, number of processors, and cost of memory are altered. In particular, we find that if the ratio of the number of processors and/or disks to the size of the database is held constant, then the cost-effectiveness of the resulting system remains constant. Furthermore, for a given size of database, there is a number of processors which optimizes cost-effectiveness. Estimated response times are also presented. Using these methods, it appears that cost-effective interactive access to databases in the 100-1000 GB range can be achieved using current technology.

1. Introduction

The study of fast, cost-effective methods for implementing IR algorithms is of great importance to their commercial usefulness. IR algorithms must be fast enough to support interactive use, and cost-effective enough that customers can afford to use them. With serial machines, achieving both speed and cost-effectiveness becomes difficult as the size of the database increases. Parallel computers offer the possibility of solving this problem, provided suitable algorithms can be found. Previous work has demonstrated a signature-based algorithm which provides interactive access to memory-resident databases, and non-interactive access to disk-resident databases. In this paper we will present an algorithm based on inverted indexes. This algorithm supports a broader range of retrieval methods than the previously published algorithm, and is suitable for access to disk-resident databases in the 100 to 1000 GB range.

Our algorithm implements document ranking using a cosine similarity measure [1][2]. As usual, we have a set of documents, each of which is represented as a vector of term weights, and a query which is also represented as a vector of term weights. The similarity of a query and a document is defined as their dot product. Retrieval consists of finding the *n* documents most similar to a given query. There are a variety of methods which may be used to determine query-term and document-term weights. As this paper is concerned with system performance rather than retrieval effectiveness, the choice of a weighting method is beyond the scope of this paper.

Currently, the usual practice for implementing a retrieval system on a serial machine [illegible] inverted indexes on secondary storage. [illegible] each document in the database is *indexed*, which reduces it to a set of terms and weights. This produces a set of *postings*, each consisting of a term, a weight, and the identifier of the document from which it was derived. This file is then sorted so that all postings for a given term are contiguous. To process a query, the postings for terms occurring in the query are first loaded into memory from disk. The postings are then used to compute a score for each document in the database, and identifiers for the highest-ranking documents are extracted. Overlap encoded signature files [3][4][5] have also been suggested as a possible organization.

In [6] Stanfill and Kahle suggested the use of the Connection Machine® System (CM), a massively parallel computer with up to 65,536 processing elements[7][8]. This paper they proposed an algorithm based on overlap-encoded signatures . Two applications of the algorithm were discussed: a batch system and an interactive system. The batch system reported ability to apply a 20,000 term query to a 15 GB database in 3 minutes. The interactive system reported searching a memory-resident database (up to 160 MB at the time the paper was written) using relevance feedback in 20 ms.

In [9] Stone argues that the batch application is inefficient for queries smaller than 20,000 terms, in the sense that an algorithm based on inverted indexes will perform much less I/O. Stone goes on to suggest the use of an inverted file structure on parallel hardware. In [10] Salton argues that 1) signature files do not support document-term weighting, which restricts them to an inferior class of weighting schemes; 2) that the CM signature algorithm is no faster than an inverted algorithm running on a Sun-4; and 3) that increases in speed over those obtainable on a serial machine are not needed. Croft [11] argues that the retrieval schemes easily implemented by signatures lose approximately 10-20% on precision at standard recall levels. In [12] Stanfill presents some benchmarks of the Sun-4 against the CM-2 (a newer model than was used in the 1986 paper), reporting an 80-fold performance advantage for the CM-2. Also, the larger memory of the CM-2 is reported as permitting databases up to 2 GB to be stored in-memory.

At present, it appears that signatures stored on secondary storage and searched sequentially are too slow to permit interactive access. Signatures stored in primary storage sacrifice some retrieval effectiveness, but appear to yield sufficient performance gains to justify their application in the interactive environment. The inverted-index algorithms presented below avoid these problems, permitting both document-term weighting and interactive access to disk-resident databases.

The organization of the paper is as follows. In section two we define our database, hardware, and cost-effectiveness models. In section three we present the parallel inverted index method for memory-resident databases. In section four we adapt the algorithm to disk-resident databases. Finally, in section five we summarize our results.

® Connection Machine is a registered trademark of Thinking Machines Corporation.

2. Preliminaries

2.1. Database Size and Structure

The size of the database in gigabytes is denoted |D|. We will consider databases of 1, 10, 100, and 1000 GB, with an average document size of 10 KB. We will consider queries of 10 terms. In cases where queuing behavior is important, query arrival will be modeled as a Poisson process, with the arrival rate being varied to demonstrate the behavior of the system as it approaches saturation.

2.2. Term Frequency Distribution

The terms contained in a query are present in the database with varying frequencies. We define n to be the number of terms in the lexicon, and $f(t_i)$ to be the frequency of the i'th term. A query is constructed by selecting some number of terms from the lexicon, according to a distribution function q (the corresponding random variable will be called T). The frequency of query-terms in the database as a whole is then the random variable $f(T)$.

We will now specify q and f. First, it has been observed that, if one ranks the words in a lexicon from most common to least common, that the product of a word's rank and its frequency in the database is nearly constant [13]. Thus, we can state $f(t_i) = c_1/i$, for some suitably chosen c_1. If N_w is the number of words in one megabyte and (as stated above) n is the size of the lexicon, we have $N_w = c_1 \sum_{i=1}^{n} \frac{1}{i}$.

Second, we have observed that the probability of a term occurring in a query is proportional to that term's frequency in the database as a whole, particularly in cases where relevance feedback is employed, except that the most common words (*stop words*) are excluded. We model this by assuming that the s most common words do not occur in queries at all. This gives us $q(i) = \begin{cases} i < s & 0 \\ otherwise & \frac{c_2}{i} \end{cases}$. Since q is a probability distribution function, we have the constraint $1 = \sum_{i=s}^{n} \frac{c_2}{i}$. Finally, we can compute the expected value of f(T) by summing $q(i)f(i)$, giving us $E(f(T)) = \sum_{i=s}^{n} \frac{c_1 c_2}{i^2}$.

In our experience, reasonable values are N_w = 125,000 and $E = 3$. Given these constraints, we can find suitable values for the other parameters: n = 200,000, s = 550, c_1 = 9778, and c_2 = .1696 (the lexicon has 200,000 words, and there are 550 stop-words). The relationship between the frequency of terms in the database vs. their frequency in queries is shown in Figure 1. As noted above, the average query-term has 3 postings per megabyte. In this plot, we see that the median query-term has 1 posting per megabyte. Thus, while common query-terms generate most of the work to be done by the algorithm, relatively rare terms make up a large portion of the workload.

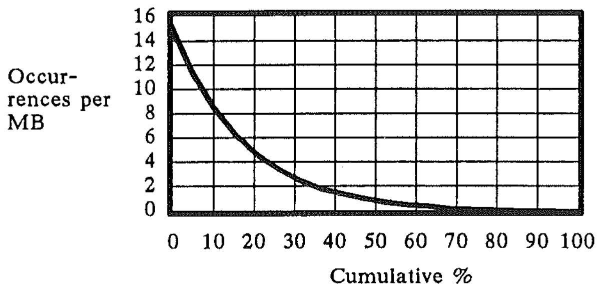


Figure 1: Term Frequency in Database vs Term Frequency in Queries

The reader may have a different view as to the shape of q and the value of E. Altering the value of E effectively scales the size of the database: a 100 GB database with $E = 3$ will behave much like a 300 GB database with E = 1. The shape of q is of more importance, but the majority of the controversy is over the shape of the curve for the most common terms: the contention is that terms of moderate frequency make up the majority of queries. However, as we will see below, it is not the intermediate terms, but the rare terms which present difficulties, and there is less controversy as to the size of the low-frequency tail.

2.3. Hardware Model

A parallel computer consists of an arbitrary number of processors connected by a communications network. Each processor has a local memory, and communicates with other processors by message passing. The computation is governed by a single program, with all processors executing the same program-step simultaneously (SIMD). Processors may be selectively activated and deactivated in order to implement conditional execution (IF statements). There are three classes of program steps: serial operations which manipulate a single datum (e.g. a loop counter); local parallel operations in which each processor operates on its own memory; and non-local parallel operations in which processors exchange information. Global and local-parallel operations include everything that serial computers can do. Non-local parallel operations are unique to parallel computers.

The basic data structure is a *parallel variable*, which can be thought of as a one-dimensional array having one data element in each processor. If the rank of a parallel variable is greater than the number of processors, then a *virtual processing* scheme may be used to simulate a machine having an arbitrary number of processors. In some cases we will distribute a 2-dimensional array across the machine such that each column of the array corresponds to a processor.

The simplest communication operation is sending, in which each processor transmits a packet of information to some other processor. In cases where multiple messages converge on one location, there are several options as to what may be done. In this paper, we will compute the sum of the data in the converging messages; this is referred to as *send-with-add*. Finally, it is also possible to determine the largest value in a parallel variable; this is referred to as *global-maximum*.

We will assume that the machine is composed of a large number of very simple processors (this assumption is not critical to the majority of our results). For such a machine, it is reasonable to assume that a 32-bit local operation takes 32 microseconds, that a global-maximum takes 100 microseconds, and that a send operation takes 1000 microseconds.

2.4. Ideal Cost Effectiveness

The *cost-effectiveness* of a retrieval system is the ratio of the size of the database to the resources expended searching it. The total resources used in a search may be computed as the *time* multiplied by the *cost* of the required hardware. The cost of the hardware will be the cost of the processor(s) plus the cost of memory. The cost of memory is proportional to the database size. This gives us:

$$CE = \frac{|D|}{T(|D|, N_p)(N_p C_p + |D| C_m)}$$

Where:

CE	=	Cost Effectiveness (GB/\$-second)
$\|D\|$	=	Size of Database in GB
Np	=	Number of Processors
Cp	=	Cost per Processor
Cm	=	Cost of storage for a 1 GB database
$T(x, y)$	=	Time to search an x GB database with a machine having y processors

Ideally, a retrieval algorithm should take time linear in the size of the database divided by the number of processors. The first condition — linearity in the size of the database — is met for indexed retrieval methods in general. The second condition — linearity of speedup — will be met by algorithms which keep processors uniformly busy. Making this assumption, we get the *ideal cost effectiveness* of searching a database (K_1 is the constant of proportionality):

$$CE_{ideal} = K_1 \frac{N_p}{(N_p C_p + |D| C_m)}$$

We may set $Cp = 1$ without altering the above relationship (K_1 will change). This leaves us with one architectural parameter - Cm - which reflects the relative cost of processors and memory. Much of this paper will be concerned with the relationship between Cm, $|D|$, and Np. For definiteness sake, we will say that $Cm = 10{,}000$ for primary storage, and 100 for disk.

Suppose now we take $Np = 1$ (the serial case). We then get $CE_{ideal} = O(1 / |D|)$. Suppose, on the other hand, we allow the number of processors to vary in proportion to the size of the database. We then get:

$$CE_{ideal} = \frac{K|D|}{K|D| C_p + |D| C_m}$$

All the |D| terms cancel out, and the cost effectiveness is constant, rather than $O(1/|D|)$. Thus, we see that the cost-effectiveness of parallel machines is asymptotically better than that of serial machines by a factor proportional to the size of the database - provided we can find an efficient algorithm.

To properly interpret the above result, we must note that the time-constants for parallel and serial implementations will be different; a fast serial processor may be two orders of magnitude faster than a single PE of a parallel machine. However no matter how much faster the serial processor is, one can find a value of $|D|$ large enough that a parallel architecture is more cost-effective.

The graph shown in Figure 2 shows ideal cost effectiveness as a function of Np, using the two values of Cm and $|D| = 1$ GB.

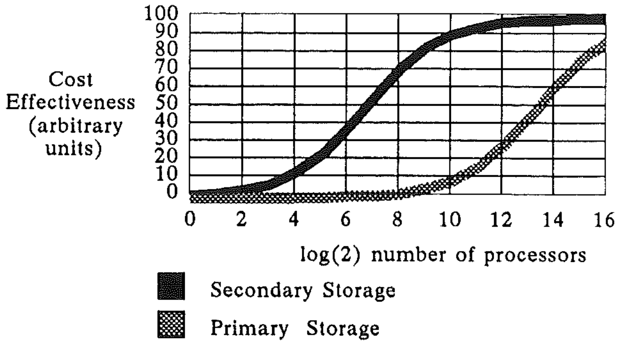


Figure 2: Ideal Cost Effectiveness
|D| = 1 GB, Primary and Secondary Storage

Cost-effectiveness rises rapidly at first, so that each doubling of the number of processors doubles the cost-effectiveness. Eventually, the point of diminishing returns is reached, and cost-effectiveness asymptotically approaches a limit we will refer to as the *optimal cost effectiveness*. Increasing the cost of storage has the effect of moving the ideal cost-effectiveness curve rightward. Increasing the size of the database has the same effect. These statements may be easily verified by examining the form of our definition of ideal cost-effectiveness or by constructing a graph.

It is interesting to consider the relationship between memory-price and optimal cost-effectiveness. On the one hand, increases in memory-price moves the ideal cost-effectiveness curve rightward, making it harder to reach the optimum. On the other hand, for a given database size, expensive memory increases the advantage of parallel computation over serial; for secondary storage, parallel computation yields approximately a 100-fold improvement in cost-effectiveness, but for primary storage the improvement is close to 10,000-fold. Similarly, for a given memory price, increasing the database size increases the number of processors required to approach the optimum, but increases the benefits of parallelism. Thus, for a 1 GB database and secondary storage, parallelism gives a 100-fold improvement in cost-effectiveness, but for a 100 GB database the improvement is 10,000-fold.

3. The Mailbox Algorithm

In this section we will present the *mailbox algorithm*, which implements document ranking for memory-resident databases. The section which follows will extend the mailbox algorithm to disk-resident databases.

In the examples in this paper, we will use the following sample database of three documents:

1:	*Information Retrieval by Parallel Document Ranking*
2:	*An Analysis of Parallel Text Retrieval Systems*
3:	*Information Retrieval in the Law Office: An Overview*

We will assign in-document weights arbitrarily.

We will evaluate the following query against the above database (again, the weights are arbitrary):

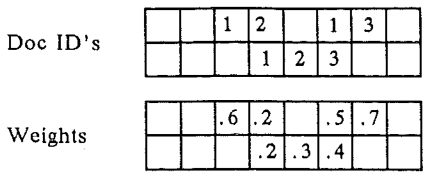
parallel * .5, information * .2, retrieval * .3

The portion of the inverted index relevant to the above database/query combination is as follows:

Word	Postings <doc-id weight>		
parallel	<1, .6>	<2, .2>	
information	<1, .5>	<3, .7>	
retrieval	<1, .2>	<2, .3>	<3, .4>

3.1. Data Representation

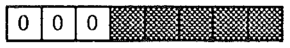
For each word which occurs in the database, we maintain a list of *postings*. The posting list for word *w* contains one posting for each document in which *w* occurs. The postings are stored in a two-dimensional parallel variable, with each column corresponding to a processor, and each row corresponding to an address in local memory. The posting-list for a given word occupies a subset of a row of this variable (this is called a *stripe*). If there are more than N_p postings for a word, then multiple stripes will be required. In most cases, all but one of these stripes will be a full row of postings. The one stripe which is not a full row is called the *fragment*. It is the size of the fragments which governs the efficiency of the mailbox algorithm. We also need to maintain an *index* which permits locating the stripes for a given word.



Word	Row	Col	Length
Parallel	0	2	2
Information	0	5	2
Retrieval	1	3	3

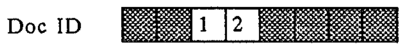
3.2. Mailboxes

For each document in the database, a 32-bit *mailbox* is allocated. The mailbox is used as an accumulator to keep track of the document's score. In our sample database of 3 documents, we have less than one mailbox per processor, but for large databases it may be necessary to put multiple mailboxes in each processor. We will refer to the first *Np* mailboxes as *row 0*, to the next *Np* as *row 1*, and so forth.

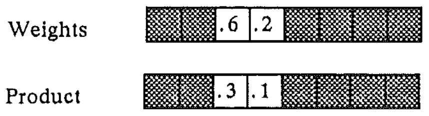


3.3. Scoring

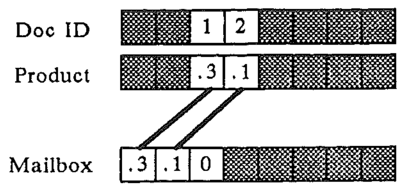
The first phase of processing computes a score for each document in the database. It is best explained by an example. Let us suppose we are processing the query term *.5 * parallel*. First, we find the index entry for *parallel*. This index entry states that *parallel* has a single stripe, which occupies processors 2 and 3 of row 0. Next, we de-select all processors which do not contain data for this stripe:



The in-document weight is next multiplied by the in-query weight (.5):



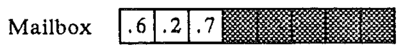
We then send the product to the mailbox addressed by the document ID; the receiving processor performs the addition.



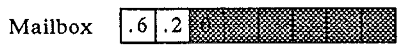
When this has been done for all query terms, the mailbox for each document contains the score for that document. In this explanation we are ignoring the complexities which arise when there are multiple rows of mailboxes. This computation is dominated by the time required to *send*; thus we see that document scoring takes approximately 1 ms per stripe.

3.4. Ranking

The final step in processing is to collect the documents having the highest scores. This can be done by repeatedly 1) finding the mailbox with the highest score; 2) deactivating that the processor containing that score. For example, assume the following scores have been computed:



Taking the global-maximum, we find that mailbox 2 (corresponding to document 3) contains .7; we deactivate that processor, leaving us in this state:



Repeating the process, we find that the new global-maximum is .6, corresponding to document 1. If we have a single row of mailboxes, we need one global-maximum operation for each document to be retrieved. There is an additional cost of 1 global-maximum for each row of mailboxes. This gives the following formula for rank-time:

$$t_{rank} = .1\left[N_{retrieve} + ceiling\left(\frac{|D|}{N_p}\right)\right] \text{ milliseconds}$$

As a note, this formula contains the constant term *Nretrieve*, and is contrary to our stipulation that retrieval take time proportional to database size and inversely proportional to the number of processors. However, for *Nretrieve* = 20, the total departure from linearity is only 2 milliseconds, and we can safely ignore the deviation.

3.5. Efficiency of the Scoring Algorithm

The only point at which processors are deactivated, hence the only point at which our algorithm can become inefficient, is during the processing of fragmentary stripes. In this section we will look at the issue of fragmentation and its effects on efficiency.

The number of stripes which must be accessed to evaluate a single query term is the random variable *ceiling*($|D|$ * $f(T)$ / N_P). For each pass through the loop the hardware has the capacity to process N_P postings. The actual number of postings moved will be $|D|$ * $f(T)$. Thus, the efficiency of the algorithm can be computed as:

$$EFF(N_p,D) = \frac{|D| * f(T)}{N_p * \text{ceiling}\left(\frac{|D| * f(T)}{N_p}\right)} \qquad (1)$$

Using the distribution of frequency terms given in Section 2.2., we can compute the average efficiency of this algorithm for databases of various sizes. The results of this computation are given in Figure 3.

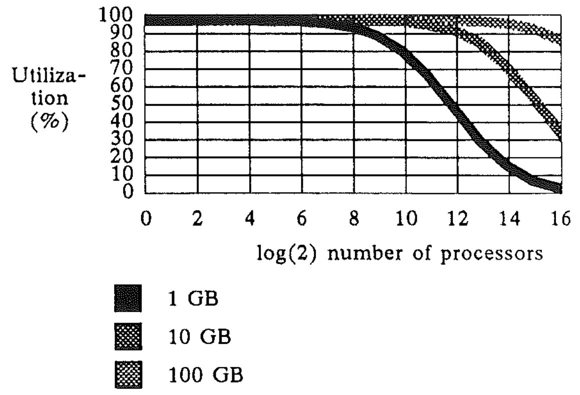


Figure 3: Efficiency of the Mailbox Algorithm

The above shows that increasing the number of processors beyond a critical region causes the efficiency of the algorithm to fall, asymptotically approaching 0. The location of this critical point depends on the size of the database and on the distribution of term frequencies $f(T)$, but does not depend on the particular machine being used. Also, one can see that increasing the size of the database has the effect of moving the curve rightward without altering its shape.

We now come to some central insights about this algorithm. We have already established that ideal cost-effectiveness starts very small and approaches a limit as the number of processors grows. We have also established that efficiency starts at 100% and decreases to 0 as the number of processors grows. Since the *actual* cost effectiveness of the algorithm is equal to the ideal cost effectiveness times the efficiency there must exist an optimal number of processors which optimizes cost-effectiveness. Consider a 1 GB memory-resident database (Figure 4).

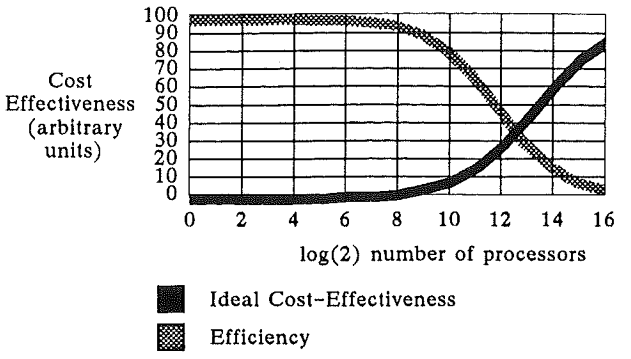


Figure 4: Ideal Cost-Effectiveness and Efficiency: 1 GB in-memory database

Evidently one wants to use 2^{12} (4096) processors, and one will attain 16% of optimal cost-effectiveness*.

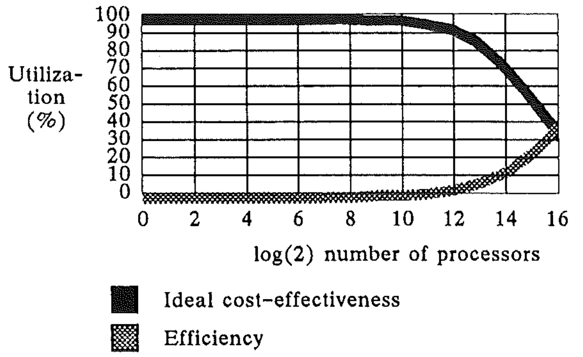


Figure 5: Ideal Cost-Effectiveness and Efficiency: 10 GB In-Memory Database

In the section on ideal cost effectiveness, we noted that increasing the size of the database shifts the ideal-cost-effectiveness curve rightward. We have just seen that increasing the size of the database shifts the efficiency curve rightward. These shifts are by identical amounts, and thus we see that the cost-effectiveness of the algorithm depends only on the architecture, and not on the size of the database. This is illustrated by the curves in Figure 5.

* While the reader may think that 16% of optimal is not very good, we must point out that the single-processor-solution (under these architectural assumptions) is .01% of optimal. In any event, we will shortly present techniques for improving on this figure.

3.6. The Buffered Mailbox Variant

We just showed that, under the architectural assumptions we have been making, the parallel in-memory algorithm achieves 16% of the optimal cost-effectiveness. In this section we will present a variant on the mailbox algorithm which uses a buffering scheme to reduce inefficiency due to fragmentation, and which comes somewhat closer to the optimum.

The only source of inefficiency in our algorithm is the inner loop of the scoring algorithm, where fragmentation leads can lead to poor processor utilization. We note that almost all the time in this algorithm is spent *send*ing data from one processor to another. If we can increase processor utilization during this *send*, we can improve efficiency.

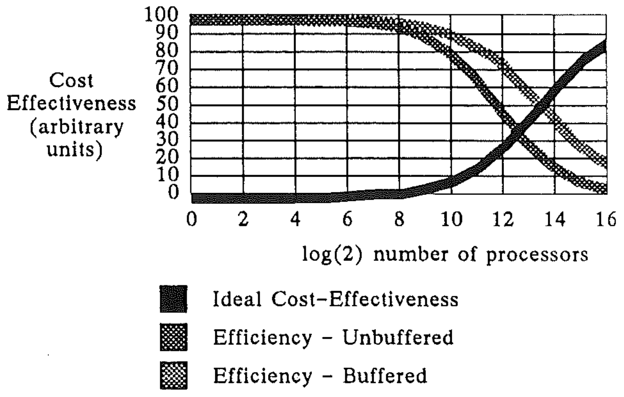


Figure 6: Ideal Cost-Effectiveness and Efficiency: 1 GB In-Memory Database

This can be done by pulling the *send* outside the inner loop. The modification to the algorithm is as follows: As the stripes for the query are processed, rather than executing a *send* instruction immediately, each active processing element will *push* a posting onto a stack. When all stripes have been processed in this manner, we can *pop* data from these stacks and perform the *send* operation. In this case, the number of *sends* required will be the length of the longest stack, rather than the number of stripes. In most cases, this number will be substantially smaller. The efficiency of this algorithm, as determined by simulation (10 term queries), is shown in Figure 6. Apparently, buffering almost doubles the cost-effectiveness of the algorithm, from 16% of optimal to 28%. More extensive use of buffering can further increase efficiency (e.g. batching together 10 queries), but doing so requires more complex software.

3.7. Overall Performance

Having done what we can to improve the efficiency of the scoring algorithm, we must now consider the overall cost of query processing, which includes both scoring and ranking. This cost of the ranking operation is significant, and 100% processor utilization is achieved during this algorithm. Thus, overall processor utilization is somewhat higher than that attained during document scoring, and cost effectiveness is improved. Simulation reveals that, for the architectural parameters we have been using, we may process a 1 GB database on 16K processors, achieving 65% processor utilization, 65% of the optimal ICE, and thus 42% of the optimal cost effectiveness (Figure 7).

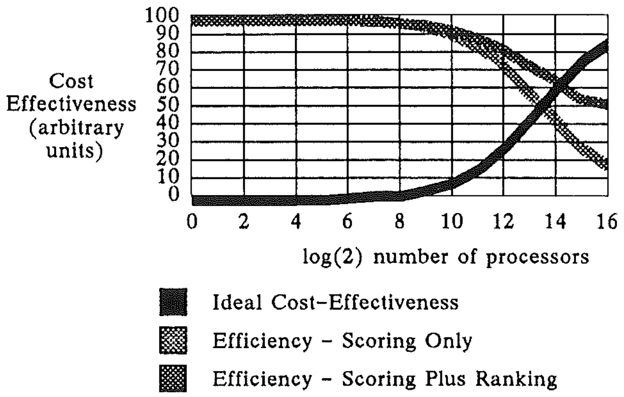


Figure 7: Ideal Cost-Effectiveness and Overall Efficiency: 1 GB In-Memory Database

It should be noted that scoring depends on *send* operations, while ranking depends on *global-maximum* operations. Altering the relative speeds of these operations will alter the relative importance of ranking and scoring to overall performance.

It is also possible to estimate times for query processing by simulating large numbers of queries. The results of this simulation (using the hardware model from Section 2.3) are shown in Figure 8. It must be emphasized that these figures depend on many assumptions: the number of terms per query, the distribution of query-term frequencies, the size of the average document, and the speeds of various hardware operations. Thus these processing time numbers are of very narrow significance. We will shortly use these times to estimate the number of processors required by disk-resident databases.

Np	1 GB	10 GB	100 GB	1000 GB
1	41	400	4000	41000
2	20	200	2000	20000
4	10	99	1000	10000
8	5.1	50	500	5000
16	2.5	25	250	2500
32	1.3	13	130	1200
64	0.6	6.3	62	620
128	0.3	3.1	31	310
256	0.2	1.6	16	150
512	0.1	0.8	7.8	77
1K	< 0.1	0.4	3.9	39
2K	< 0.1	0.2	2.0	19
4K	< 0.1	0.1	1.0	9.7
8K	< 0.1	< 0.1	0.5	4.9
16K	< 0.1	< 0.1	0.3	2.5
32K	< 0.1	< 0.1	0.1	1.2
64K	< 0.1	< 0.1	< 0.1	0.6

Figure 8: Estimated Processing Times (sec), **10-Term Queries**

4. The Disk-Resident Database

In this section we will consider issues which arise in conjunction with disk-resident databases. The characteristics of disks are rather different than those of primary storage, so the cost-effectiveness and efficiency issues which arise are somewhat different.

We make the following assumptions about the disk system. A machine may have an arbitrary number of direct access devices. An I/O operation consists of reading one block of information (maximum size 256Kbytes) from each drive. The data from each drive is written to a single parallel variable, so that if we read 256 Kbytes from each of 32 drives into a 64K processor machine, each processor will receive 4 bytes of information from each of the 32 drives. The average latency (rotational + seek) is 30 milliseconds, and the transfer rate is 1 MB/second. A disk holds 1 GB data, which is sufficient to store the postings for a 3 GB database.

The primary architectural variable here is the size of the disks. Given a fixed database size, the number of disks is thus determined. The number of disks determines the number of I/O's per second which may be performed, and the maximum transfer rate. These, in turn, determine the maximum load which can be sustained by the disk system. Given this number, one needs only sufficient processors to keep the disks busy. Thus, a 10 GB database will fit on 3 disks; a 100 GB database will fit on 33 disks; and a 1000 GB database will fit on 333 disks. We will now proceed to establish the circumstances under which these disks may be efficiently utilized.

4.1. The Basic Algorithm

The organization of the database on disk is similar to that used for databases stored in primary memory. The disk is divided into 256K-byte blocks, each of which holds 64K postings. The postings for a word are stored in as many blocks as are needed, on consecutive disk drives. As was the case with the in-memory algorithm, a certain amount of fragmentation is inevitable. There is an index which allows the postings for a given word to be located on disk.

Processing consists of determining the disk-address of the postings for a word, loading the postings into memory, then using the in-memory mailbox algorithm to process the postings.

The algorithm, as outlined above, suffers from inefficiency due to load imbalance. Suppose, for example, that a query requires 256 blocks from disk, and that the system has 256 disks. The block requests will be scattered among the 256 disks in the system more-or-less at random, with the effect that some drives will fetch no data, while one drive might need to fetch 4 blocks of data. The result is that 4 cycles of direct access I/O would be required. Thus, in the time when we might have been reading 1024 disk blocks, we will actually read only 256, and we are getting only 25% utilization out of our disk system. The actual magnitude of the imbalance effect may be determined by simulation using the term-frequency distribution given earlier:

DB Size	Disks	Utilization
10 GB	3	50%
100 GB	33	37%
1000 GB	333	37%

4.2. Shared Disk Queues

Better disk utilization may be attained by creating a shared disk queue along the following lines. A query is executed in three phases. In the first phase, the terms are extracted from the query and a list of all disk blocks required for the query is compiled. The addresses of the required disk blocks are then given to a disk handler, which maintains a queue of requests for each disk in the system. Processing for that query will then be suspended. When all data blocks have been read, execution of the query will be resumed and the in-memory algorithm executed as described above. Concurrently, other queries could arrive in the system, generating their own sets of disk requests. Suppose we have two disks, and two queries A and B which arrive in the system at the same time. Furthermore, suppose A requires two blocks from drive 1, and B requires 1 blocks from each drive. Clearly, if we execute the two queries independently, we achieve only 50% disk utilization, but if we pool disk requests we achieve 67% disk utilization. By pooling the requests for a sufficiently large number of queries, we can keep the disks arbitrarily active, although at the cost of increased response time.

4.3. Throughput

The degree of disk utilization we can actually achieve is limited by queuing delays: like any other queuing system, as utilization of system resources approaches 100%, queuing delays approach infinity. In order to understand this effect, we constructed a queuing model using the system parameters described above. We then simulated the system with a variety of query arrival rates. Figure 9 shows the result of this simulation. For a system with 32 disks and a 100 GB database, the ultimate capacity of the system can be estimated at 1.8 queries/second, at which point queuing delays would be quite large. However, with an arrival rate of 1.4 queries/second, the average response time is a reasonable 3.0 seconds, and the system is operating at 77% of capacity.

Arrival Rate (Queries/sec)	Response Time (sec)	Disk Utilization (%)
.2	1.1	11%
.4	1.2	22%
.6	1.3	33%
.8	1.5	43%
1.0	1.8	54%
1.2	2.3	66%
1.4	3.0	77%
1.6	5.3	87%

Figure 9: 100 GB disk-resident database, shared disk queue algorithm (32 disks)

If we are to handle 1.4 queries per second, we need sufficient processors to evaluate an average query in .7 seconds. Referring back to Figure 8, we see that 8K processors are required to achieve .5 second response on a 100 GB database. Using the efficiency model previously explained, we find that this system gives us an efficiency of near 100%. Using the cost effectiveness formula with $Cm = 100$ (disk), the cost effectiveness of this configuration is 45% of optimum.

Thus, for a 100 GB database resident on external storage, we can achieve high resource utilization and near-optimal cost effectiveness with 32 disks and 8K processors. The resulting system can comfortably process 1.4 queries per second with a response-time of 3.5 seconds. As was the case with all other aspects of the mailbox algorithm, the solution is stable as the system scales. If we repeat these simulations, we find that a 1000 GB database requires 333 disks, 64K processors, and will achieve similar response times and cost-effectiveness.

5. Conclusions

In summary, we have proposed a parallel algorithm for IR by document ranking, which uses inverted indexes. We have established its computational properties as follows:

1. The parallel algorithm's cost-effectiveness is constant, while the serial inverted index's cost-effectiveness is inversely proportional to the size of the database.
2. For any size database, there is a number of processors which optimizes cost-effectiveness. This number depends on the relative cost of processors and storage.
3. The attainable cost-effectiveness is stable as the size of the database increases.
4. An efficient strategy for using the algorithm with secondary storage exists.
5. Using current technology, this algorithm permits interactive access to databases up to 1000 GB.

REFERENCES

[1] Van Rijsbergen, C.J., *Information Retrieval, Second Edition*, Butterworths, London, 1979.

[2] Salton, G., *The SMART Retrieval System — Experiment in Automatic Document Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[3] Christodoulakis, S. and Faloutsos, C., "Design considerations for a message file server," *IEEE Transactions on Software Engineering*, Volume SE-10, 1984, pp. 201-210.

[4] Faloutsos, C. and Christodoulakis, S., "Signature files: An access method for documents and its analytic performance evaluation," *ACM Transactions on Office Information Systems*, Volume 2 Number 4, October 1984, pp. 267 - 288.

[5] Tsichritzis, D. and Christodoulakis, S., "Message files," *ACM Transactions on Office Information Systems*, Volume 11 Number 1, January 1983, pp. 88-98.

[6] Stanfill, C. and Kahle, B., "Parallel Free-Text Search on the Connection Machine System," *Communications of the ACM*, Volume 29 Number 12, December 1986, pp. 1229 - 1238.

[7] Hillis, D., *The Connection Machine*, MIT Press, Cambridge MA, 1985.

[8] "Connection Machine® Model CM-2 Technical Summary, Technical Report HA87-4, Thinking Machines Corporation, Cambridge MA, 1987.

[9] Stone, H., "Parallel Querying of Large Databases: A Case Study," *IEEE Computer*, October 1987, pp. 11-21.

[10] Salton, G. and Buckley, C., "Parallel Text Search Methods," *Communications of the ACM*, Volume 31 Number 2, February 1988, pp. 202-215.

[11] Croft, Bruce, "Implementing Ranking Strategies Using Text Signatures," *ACM Transactions on Office Information Systems*, Volume 6 Number 1, January 1988, pp. 42-62.

[12] Stanfill, C., "Parallel Computing for Information Retrieval: Recent Developments," Technical Report DR88-1, Thinking Machines Corporation, Cambridge MA, January 1988.

[13] Zipf, H.P., *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Cambridge MA, 1949.